05｜跨越现实的障碍（中）：富含知识还是代码坏味道？

2021-07-03 徐昊

《如何落地业务建模》 课程介绍 >

讲述：徐昊
时长 22:05 大小 20.23M

你好，我是徐昊。今天我们来聊聊如何有效地分离领域模型中的上下文。

上节课我们介绍了关联对象，以及如何通过关联对象建模聚合 / 关联关系，从而在不影响性能的情况下，更好地维持领域模型与软件实现的统一。

今天，我们来看另一个在关联领域模型与软件实现时经常遇到的问题：上下文过载（Context Overloading）。所谓**上下文过载**，就是指**领域模型中的某个对象会在多个上下文中发挥重要作用，甚至是聚合根**。一来这个对象本身会变得很复杂，造成模型僵化；二来可能会带来潜在的性能问题。

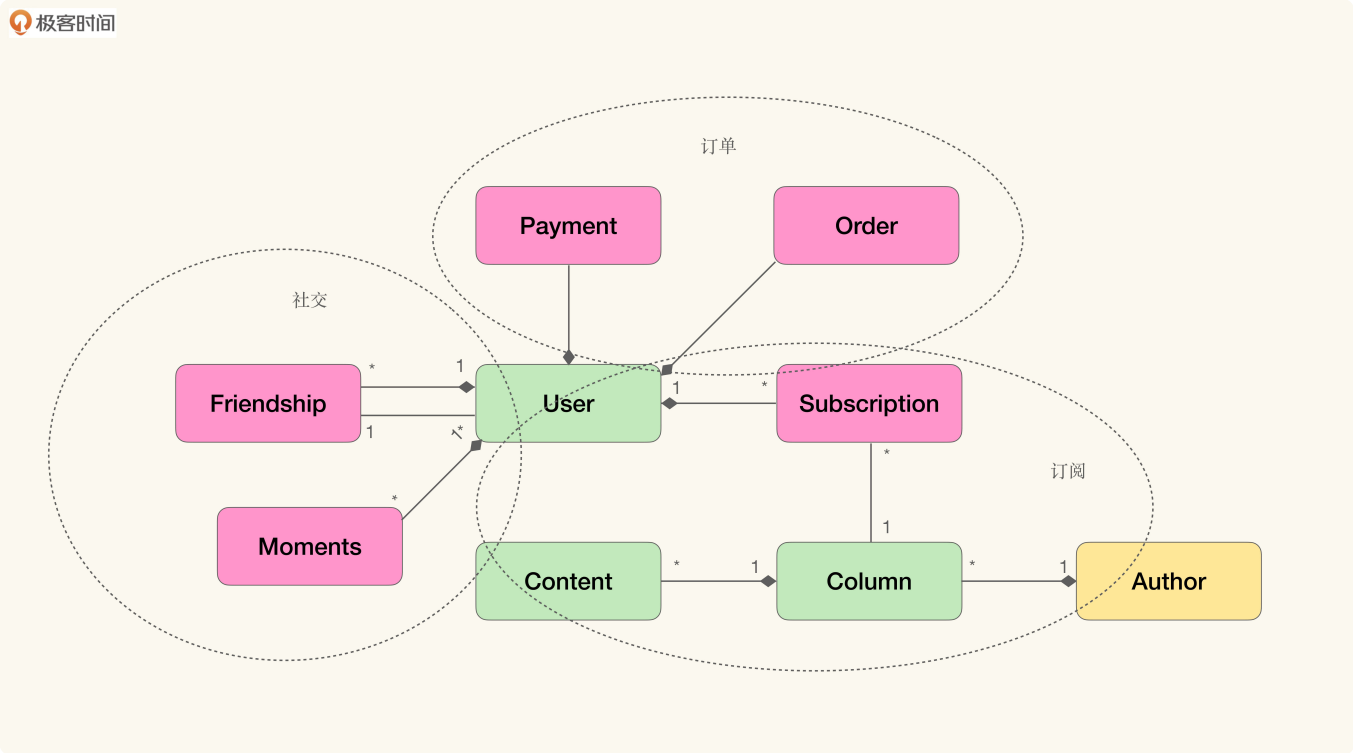
于是，将过载的上下文进行有效的分离，就成了我们现在需要面对的问题。方法上也有很多不同的选择，我主要介绍**角色对象**（Role Object）和**上下文对象**（Context Object）这

两种。这两个方法与我们上节课讲的关联对象一脉相承，都是通过将隐含概念显式建模来解决问题。

同样地，在看到具体的解决方法之前，我们需要深入了解上下文过载会产生哪些问题。

因富含逻辑而产生的过大类

为了帮助你理解上下文过载会产生哪些问题，我还是借助极客时间专栏的例子来说明一下。当然，首先我们需要扩展这个领域模型。



在这个扩展了的模型中，包含了三个不同的上下文。

1. 订阅：用户阅读订阅内容的上下文，根据订阅关系判断某些内容是否对用户可见；
2. 社交：用户维护朋友关系的上下文，通过朋友关系分享动态与信息；
3. 订单：用户购买订阅专栏的上下文，通过订单与支付，完成对专栏的订阅。

按照这个模型，我们很容易得到与之对应的“富含知识的模型”：

复制代码

```
1 public class User {
2     private long id;
3
```

```
4
5       // 社交上下文
6       private List<Friendship> friends;
7       private List<Moments> moments;
8
9       // 订阅上下文
10      private List<Subscription> subscriptions;
11
12      // 订单上下文
13      private List<Order> orders;
14      private List<Payment> payments;
15
16      // 社交上下文
17      public void make(Friendship friend) {
18        ...
19      }
20
21      public void break(Friendship friend) {
22        ...
23      }
24
25      // 订单上下文
26      public void placeOrder(Column column) {
27        ...
28      }
29
30      // 订阅上下文
31      public boolean canView(Content content) {
32        ...
33      }
    }
```

如果对代码坏味道敏感的同学，估计已经发现问题所在了：**一个对象中包含了不同的上下文**，而这恰好是**坏味道过大类**（Large Class）的定义。

那么过大类会带来什么问题呢？首当其冲是**模型僵硬**。想要理解这个类的行为，就必须理解所有的上下文。而只有理解了所有的上下文，才能判断其中的代码和行为是否合理。

于是，**上下文的过载就变成了认知的过载**（Cognitive Overloading），**而认知过载就会造成维护的困难**。通俗地讲，就是**“看不懂、改不动”**，代码就变成“祖传代码”了。

但是我们不要忘了，这是与模型关联的代码啊！**改不动的代码就是改不动的模型**！改不动的僵硬的模型，要怎么参与提炼知识的循环呢？

当然，过大类还容易滋生重复代码、引入偶然耦合造成意外的缺陷等编码上的问题。不过这些问题与模型僵硬比起来，似乎有点不值一提。

另一个问题没那么严重，但同样值得重视，这个问题仍然与性能有关。在不同的上下文中，我们需要访问的数据也不尽相同。

比如在维护朋友关系、发布动态信息时，我们不太可能访问订阅与订单信息。类似的，当我们访问订单信息的时候，也不太可能关注朋友信息。那么，当所有的上下文都汇聚于 User 时，我们怎么才能获得一个高效的聚合加载策略呢（当然通过关联对象我们能缓解这个问题，具体请看上一课）？

这里的吊诡之处在于，我们并没有做什么坏事，反而是因为严格遵照“富含知识的模型”的原则，才导致了上下文过载。

我们来分析一下。User 在不同的上下文中都处在聚合根的位置：

在订阅上下文中，User 是 Subscription 的聚合根；

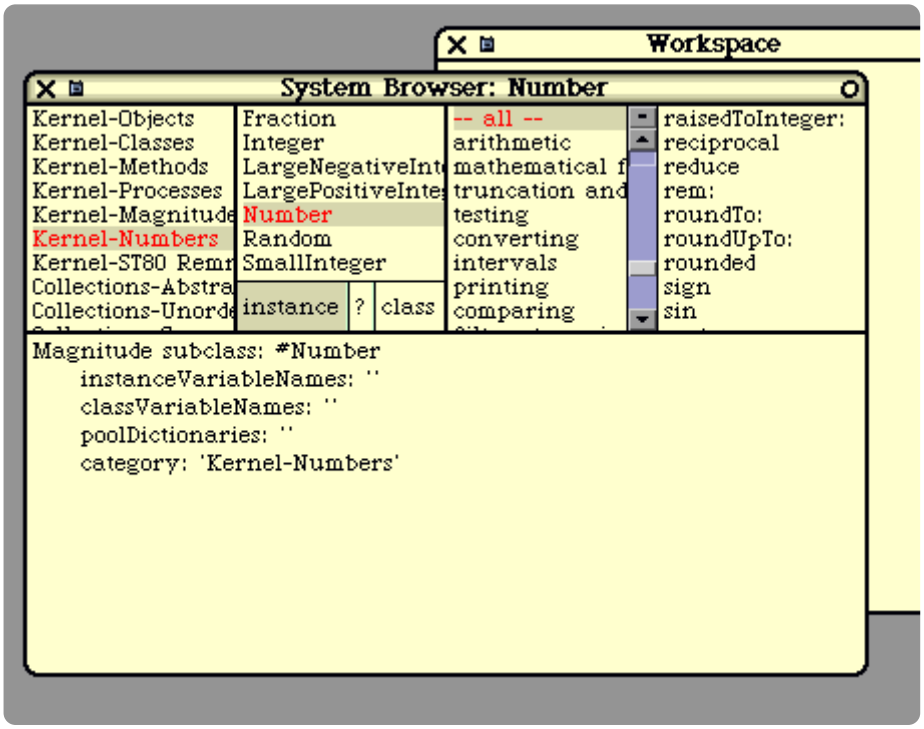
在订单上下文中，User 是 Payment 和 Order 的聚合根；

在社交上下文中，User 是 Friendship 和 Moments 的聚合根。

根据构建“富含知识模型”的做法，被聚合对象的集合逻辑放置于聚合 / 聚合根里，而不是散落在外，或是放在其他无关的服务中。这么做可以使得逻辑富集于模型中，同时有利于通过聚合关系管理一组对象与领域概念的映射，保持领域概念与逻辑的完整性。

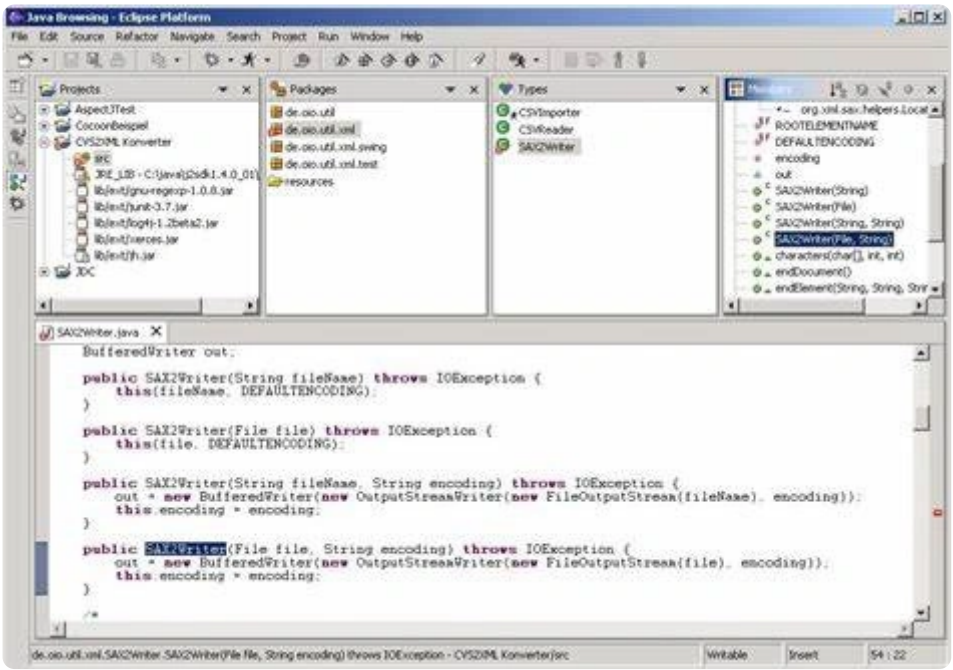
那么，每当 User 在不同的上下文中处于聚合 / 聚合根位置的时候，与之相关的逻辑都会进入 User 中，User 会不可避免地膨胀。这就是上下文过载。

又要再说一句题外话了。在使用原味面向对象范型（Vanilla Object Oriented）的时候，不自觉地就会产生过大类，这其实也跟 Smalltalk 有关。正如我们上节课所讲，Smalltalk 是一个面向对象系统，其中还包含一个 IDE：**System Browser**。



在 Smalltalk 中，源代码并不以文本格式作为主要表达形式，而是通过一个图形化的界面，让我们有指导性地、片段式地浏览对应的代码。上图中左起第三个窗口就是自定义的上下文。也就是说，即使对象中存在再多的上下文，也不会影响我们对代码的理解和修改。毕竟上下文再多，我们也不会一次看到多个方法的具体实现代码。

如果你用过 Eclipse IDE 开发 Java 项目的话，那么这里肯定有一个不那么有用的古怪的视图，让你百思不得其解。但如果你对这张图和上面 Smalltalk System Browser 有点了解的话，就能从布局中看出一些端倪。

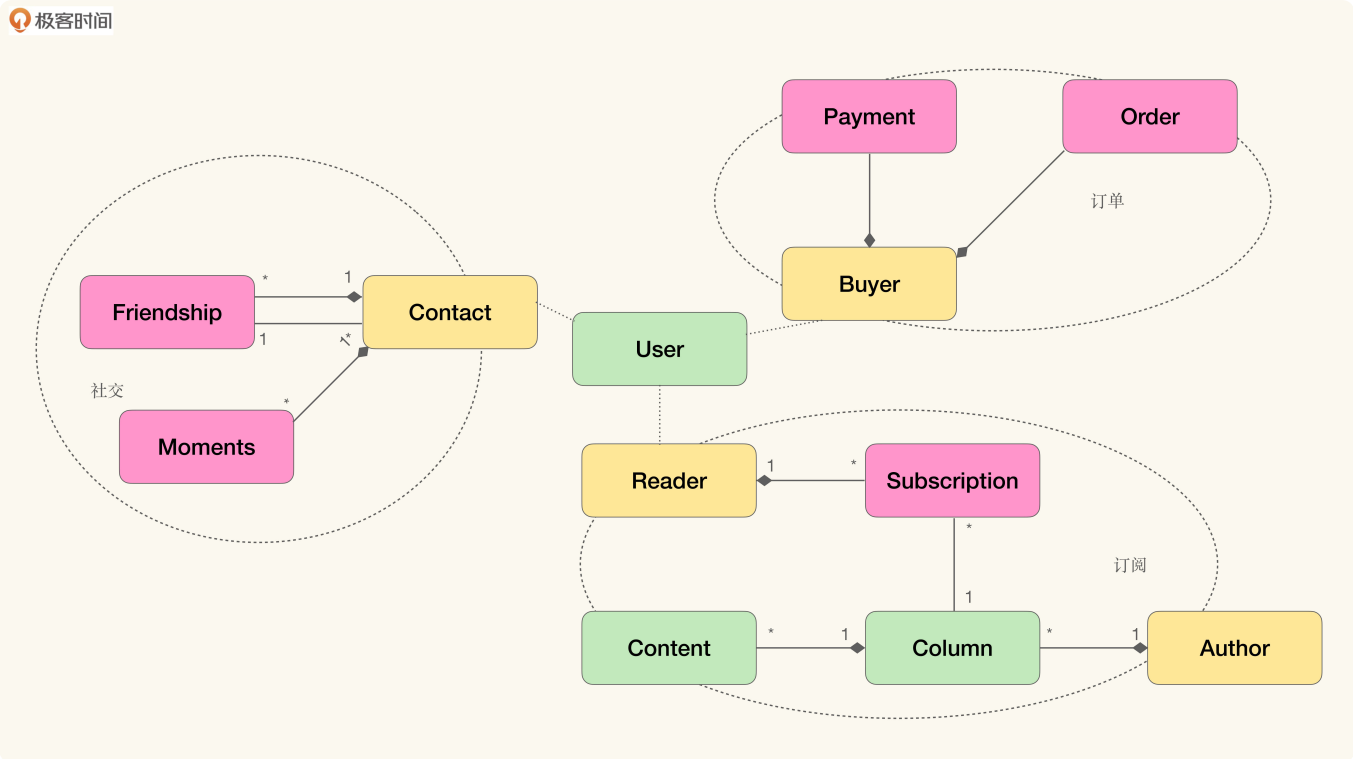


这其实是 Eclipse 对 Smalltalk 的致敬。毕竟，Eclipse 也算是传奇 Smalltalk 程序员 Kent Beck 的作品。但当我们以文本作为源代码的主要形式，以类作为源代码的主要组织方式时，仅仅通过视图的切换是无法达成 Smalltalk 的效果的。于是，过大类的上下文过载就成了一个严重的问题。

逻辑汇聚于上下文还是实体？

上下文过载问题最根本的症结在于，**逻辑是需要汇聚于实体**（User）**还是上下文**（订阅、社交与订单）。

原味面向对象范型（也是领域驱动设计的默认风格）的答案是汇聚于实体，但是缺少有效分离不同上下文的方式。而 DCI 范型（Data-Context-Interaction，数据 - 上下文 - 交互）要求汇聚于显式建模的上下文对象（Context Object），或者上下文中的角色对象（Role Object）上。如果按照 DCI 范型的思路，我们可以如下图这样建立模型：



这其实也很容易理解。**在不同的上下文中，用户是以不同的角色与其他对象发生交互的，而一旦离开了对应的上下文，相关的交互也就不会发生了。**

这与我们生活中的体验是一致的。上班时，以员工的身份发生的工作活动与交互，下班之后其实可以不用发生了。因为已经脱离了工作这个上下文，不再扮演这个角色了。同样的，在生活中的以其他身份发生的活动与交互，在工作中也不该发生。

也就是说，在订阅上下文中的读者（Reader），不会以读者的身份与订单上下文中的支付信息发生交互。而买家（Buyer），也不会以买家的身份去社交上下文中去建立朋友关系。

从 DCI 的角度看待聚合与聚合根关系，我们可以发现，并不是 User 聚合了 Subscription，而是订阅上下文中的 Reader 聚合了它。同时，并不是 User 聚合了 Friendship 与 Moments，而是社交上下文中的 Contact 聚合了它们。可以说，**User 只是恰好在不同的上下文中扮演了这些角色而已。**

理清了这一层关系，我们也就发现了上下文过载的根源：**实体在不同的上下文中扮演的多个角色，再借由聚合关系，将不同上下文中的逻辑富集于实体之中，就造成了上下文过载。**

明白这一点，解决方案也就呼之欲出了：针对不同上下文中的角色建模，将对应的逻辑富集到角色对象中，再让实体对象去扮演不同的角色，就能解决上下文过载的问题了。

当然，理想永远是清晰且美好的，在实践中却没有这么简单，主要是**如何在代码中实现这种实体与角色间的扮演关系。**

通过角色对象分离不同上下文中的逻辑

一个最简单的想法是，将角色抽象成接口：

复制代码

```
//阅读上下文
interface Reader {
    List<Subscription> getSubscriptions();

    boolean canView(Content content);
}


//订单上下文
interface Buyer {
    List<Order> getOrders();

    List<Payment> getPayments();

    void placeOrder(Column column);
}

```

```
//社交上下文
interface Contact {
    List<Friendship> friends;

    List<Moments> moments;

    void make(Friendship friend);

    void break(Friendship friend);
```

然而，这实际上并没有解决什么问题：

复制代码

```
public class User implements Reader, Buyer, Contact {
    // 社交上下文
    public void make(Friendship friend) {
      ...
    }

    public void break(Friendship friend) {
      ...
    }

    // 订单上下文
    public void placeOrder(Column column) {
      ...
    }

    // 订阅上下文
    public boolean canView(Content content) {
      ...
    }
}
```

通过实现接口去完成角色扮演，得到的代码其实没什么差别。虽然借助当代 IDE，我们可以较容易地判断行为来自哪个上下文，但实际得到的改善极其有限。既然接口不好用的话，另一个思路就是通过**装饰器**（Decorator），我们可以构造一系列角色对象（Role Object）作为 User 的装饰器：

复制代码

```
public class Reader {
    private User user;
    private List<Subscription> subscriptions;

```

```
    public Reader(User user) {
        ...
    }

    // 订阅上下文
    public boolean canView(Content content) {
      ...
    }
}


public class Buyer {
    private User user;

    private List<Order> orders;
    private List<Payment> payments;

    public Buyer(User user) {
        ...
    }

    // 订单上下文
    public void placeOrder(Column column) {
      ...
    }
}


public class Contact {
    private User user;

    private List<Friendship> friends;
    private List<Moments> moments;

    public Contact(User user) {
        ...
    }

    // 社交上下文
    public void make(Friendship friend) {
      ...
    }

    public void break(Friendship friend) {
      ...
    }
```

我们很容易地将不同上下文中的行为与逻辑富集到对应的角色对象中，并且与领域模型中的概念完全对应起来。然而这里还有最后一个问题，就是**如何构造这些装饰器对象。**

从代码中我们可以看到，除了行为之外，我们将对应的聚合与关联关系也都富集于角色对象里了，而这些被聚合的对象极大可能是被持久化在数据库中的数据。所以，简单的调用构造函数，并不能保证我们能从数据库中有效地读取这些数据：

复制代码

```
User user = repository.findById(....);


Buyer buyer = new Buyer(user); // 内存中构造角色对象
Reader reader = new Reader(user); // 并不会从数据库中读取
Contact contact = new Contact(user); // 其上的关联关系
```

所以，我们需要将构造角色对象的逻辑放置在能够触发技术调用的地方。那么最自然的一个选择，就是 Repository 对象了。

复制代码

```
public interface UserRepository {
    User findById(long id);

    Buyer asBuyer(User user);

    Reader asReader(User user);

    Contact asContact(User user);
}


public class UserRepositoryDB implements UserRepository {
    public User findById(long id) {
        return db.executeQuery(...)
    }

    public Buyer asBuyer(User user) {
        return new Buyer(user, db.executeQuery(...));
    }

    public Reader asReader(User user) {
        return new Reader(user, db.executeQuery(...));
    }

    public Contact asContact(User user) {
        return new Contact(user, db.executeQuery(...));
    }
}
```

于是，我们获取角色对象的方法也就可以变成：

```
User user = repository.findById(....);


Buyer buyer = repository.asBuyer(user);
Reader reader = repository.asReader(user);
Contact Contact = repository.asContact(user);
```

到此为止，通过角色对象我们得到了还不错的结果——几乎就是完美的方案了。这个方案的优势有以下几点。

首先，我们**把不同上下文中的逻辑分别富集于不同的角色对象中**。

当我们需要改动代码时，只需要找到对应上下文中的角色对象，在角色对象中进行修改即可。这一方面解决了认知过载的问题，避免了模型僵化。另一方面，因为角色对象的封装，代码的改变并不会进行传播。

比如如果我们需要修改订阅上下文中的逻辑，只需找到与订阅上下文相关的角色对象 Reader 就行了。因为我们只改动了 Reader，而 User、Buyer、Contact 都没有任何变化，所以这个修改没有传播到其他的上下文中。

与之对比的是，如果没有角色对象，我们就需要在 User 中修改，那么修改就会被传播到所有的上下文中。

其次，**从实体对象转化到角色对象经由了显式的方法调用**（asBuyer、asReader、asContact），**这实际上表示了上下文切换**。

也就是说，当我们调用 asBuyer 时，我们的上下文就从无限定上下文，切换到了订单上下文。那么在这个上下文中，我们对影响性能的集合加载就有了更明确的答案（比如订单历史预加载、支付历史延迟加载等等），进一步避免因为性能问题，破坏模型与软件实现的一致性。

不过你应该注意到了，我刚才说这“几乎是完美的方案”。那是因为我们还可以**在揭示意图与技术解耦上做得更好**。

通过上下文对象分离不同上下文中的逻辑

如同我们之前在关联对象中讲到的，获取关联集合的方式可能是异构的，也就是不止从数据库中读取这一种方法。如果社交上下文中的朋友关系，是通过服务接口调用，从其他社交平台获取的呢？这个时候，我们将 asContact 方法置于 UserRepositoryDB 之内，就显得不那么合适了。

我们来换个思路。我们可以遵循关联对象的思路，将上下文直接建模出来，并通过接口隔离具体实现：

复制代码

```
interface SubscriptionContext {
    interface Reader {
        boolean canView(Content content);
    }

    Reader asReader(User user);
}


interface SocialContext {
    interface Contact {
        void make(Friendship friend);
        void break(Friendship friend);
    }

    Contact asContact(User user);
}


interface OrderContext {
    interface Buyer {
        void placeOrder(Column column);
    }

    Buyer asBuyer(User user);
```

那么，我们就可以将上下文对象的获取放置在 UserRepository 上，对它进行改写：

复制代码

```
interface UserRepository {
    User findUserById(long id);

    SubscriptionContext inSubscriptionContext();
    SocialContext inSocialContext();
    OrderContext inOrderContext();
}


public class UserRepositoryDB {
    //通过依赖注入获取不同的上下文对象
    @Inject private SubscriptionContext subscriptionContext;

    @Inject private SocialContext socialContext;

    @Inject private OrderContext orderContext;

    ...
}

最后的使用方式就成了：

UserRepository users = ....;


User user = users.findUserById(...);


Buyer buyer = users.inOrderContext().asBuyer(user);
Reader reader = users.inSubscriptionContext().asReader(user);
Contact contact = users.inSocialContext().asContact(user);
```

现在这就是完美的方案。通过增加上下文对象，我们获得了诸多好处。

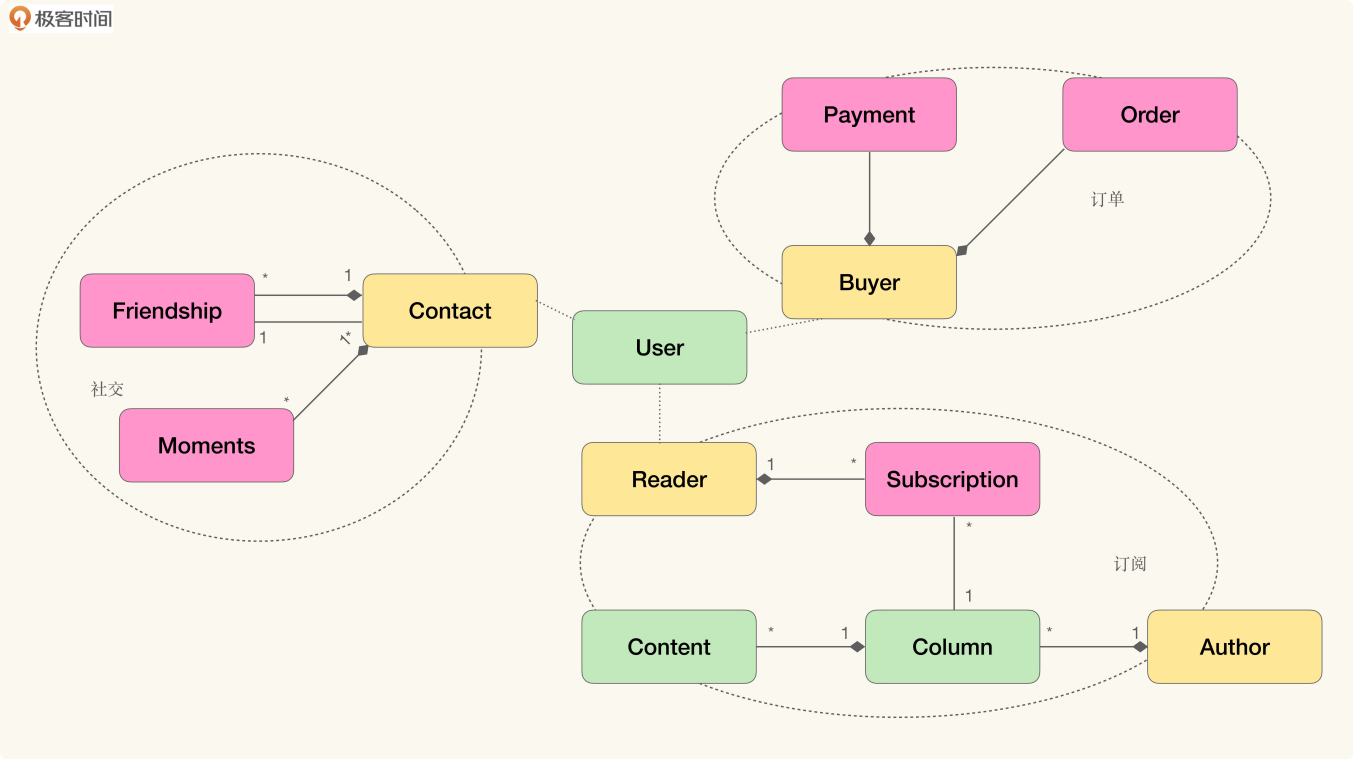
第一，**借由上下文对象的封装，不同上下文中的技术可以是完全异构的**。也就是在不同的上下文中，我们可以使用不同的技术方案。而这些技术的细节，不会暴露给外在的使用者。

比如订阅上下文可以是数据库的，社交上下文是通过三方 API 与外界服务整合的，订单上下文则是通过消息系统（Messaging System）与企业内进销存系统整合的。

第二，**软件实现、模型与统一语言更加紧密地关联在了一起。**

我们知道界限上下文的名字是统一语言的一部分。在模型中虽然不以实体的形式出现，但总归还是有它的标识（虚线的上下文边界）。如果没有上下文对象，那么它不会是软件实现的一部分，总是觉得不够理想。而有了上下文对象，统一语言、领域模型和软件实现就彻底一致了。这种舒服的滋味实在难以言表啊！

第三，**我们更加清楚地揭示了领域知识的意图**。特别是在对照这领域模型，参看 Repository 接口的时候：



复制代码

```
interface UserRepository {
    User findUserById(long id);

    SubscriptionContext inSubscriptionContext();
    SocialContext inSocialContext();
    OrderContext inOrderContext();
}
```

通过这个定义，我们清晰地知道，User 在三个不同的上下文中扮演不同的角色。而没有上下文对象的时候，这些信息则没有被披露，会增加理解的成本和负担。

上下文间的依赖关系

不过这里还有一个问题，**上下文对象在处理复杂问题的时候，还能继续保持简洁和优雅吗？**让我们再来看一个略微复杂的问题。比如我们要做个新业务，将我所订阅的专栏赠送给我的朋友。

直觉告诉我们，这需要跨越两个不同的上下文，才能构建这个业务。也就是说，在这个业务中，处在订阅上下文中的 Reader，也是处在社交上下文中的 Contact。单纯从实现它来说，问题也不难解决：

复制代码

```
User user = users.findUserById(...);
User friend = users.findUserById(...);


Contact contact = users.inSocialContext().asContact(user);


Reader reader = users.inSubscriptionContext().asReader(user);
Subscription subscription = reader.getSubscriptions(...);


if(contact.isFriend(friend)) {
    reader.transfer(subscription, friend);
```

这里的问题是，“只有朋友间才能赠送”属于领域逻辑。我们希望它被富集到模型中，而不是在模型外，那么我们要怎么处理呢？答案出乎意料得简单，在上下文对象中加入依赖就好了：

复制代码

```
public class SubscriptionContextDB implements SubscriptionContext {
    @Inject SocialContext SocialContext;

    public Reader asReader(User user) {
        //在这里将依赖的上下文传到角色对象中去就好了
    }
}
```

而实现这个业务代码就成了：

复制代码

```
User user = users.findUserById(...);
```

```
User friend = users.findUserById(...);


Reader reader = users.inSubscriptionContext().asReader(user);
Subscription subscription = reader.getSubscriptions(...);


reader.transfer(subscription, friend)
```

通过显式建模上下文对象，我们不光将业务逻辑富集到领域模型中，而且在跨域多个上下文实现某个业务时，上下文之间的依赖关系还成了实现细节，被封装于上下文对象中了。多么美妙啊！

小结

我们来简单地总结一下。当我们严格遵照“富含知识的模型”的原则，将聚合关系有关的逻辑富集到领域模型上时，很容易产生上下文过载的问题，其根源在于实体在不同的上下文中扮演了多个角色。

而上下文过载会导致模型僵化，也就是“看不懂改不动”的代码变成了“看不懂改不动”的模型，这样提炼知识的循环就无法展开了。因而我们需要将上下文从领域对象上剥离，解决办法就是角色对象和上下文对象。

需要强调的是，虽然仅仅使用角色对象也能解决问题，但是配合上下文对象一起，它们能在揭示意图的同时，使模型、统一语言与软件实现更紧密地关联在了一起，是非常有用的模式。

思考题

关联对象、角色对象和上下文对象都属于结构模式（Structural Pattern），帮助我们组织对象中的数据。那么从行为模式（Behavioral Pattern）上看，我们还会遇到什么问题呢？

欢迎把你的思考和想法分享在留言区，我会和你交流。同时呢，我也会把其中一些不错的回答置顶，供大家学习。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 3　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　04 | 跨越现实的障碍（上）：要性能还是要模型？

下一篇　06 | 跨越现实的障碍（下）：架构分层就对了吗？

精选留言 (21)

写留言

阿鸡
2021-07-03

魔鬼都在细节中，尝试去落地DDD的人多多少少都会遇到今天文章中说的问题。
从问题出发，一路分析到最终的解决方案，可见老师功力之深。
读下来畅快淋漓，爽！

13

Jxin
2021-07-07

1.徐昊老师的代码看起来很舒服啊。就是喜欢看这种面相意图的的代码，看代码就像在看故事。

2.关于上下文，徐昊老师的这个写法我是第一次见，很有意思，长见识了。我自己一般是将上下文作为一个《应用层》到《领域层》中间的层级（乐衷于加层），用于区分不同...

展开

作者回复: 扮演并不是派生关系 之所以不把角色做成聚合根 因为它们不是独立的对象 不会用reader buyer表 数据上都是user表。所以并不是微信和微信视频号的关系。那是关联而不是扮演。分包除非有arch unit去加测试 否则不如对象约束力强。最后 简单与否在于是否与概念中的模型一致，而不在于代码的写法

1 4

fzhichao

2021-07-18

如果别的语言呢，比如ruby 通过module或者rails的concern来拆解，感觉concern是不是同样的意图（asReader 或者 asBuyer）？当然单纯的concern是没有办法强制隔离的（你依然可以把代码混在一起写）也许再加上rails engine 拆开？（subscription、social等）然后通过unit test保证每个组件都是可以单测无依赖的（如果tdd的话，那肯定就可以保证了）

展开

作者回复: 我觉得scala的implicit conversation最自然。ruby include module 表示context也可

勿更改任何信息

2021-07-18

最后一个赠送的举例没明白，朋友才能赠送的规则在哪里体现了？

邓志国

2021-07-18

把社交的上下文传递到reader里，是不是让reader耦合了社交呢？

作者回复: 实现随便耦合 接口看不出来就行 耦合不是坏事 不当耦合才是

卓然
2021-07-17

不要user模型，直接建模reader,buyer,contact不行吗?

展开

作者回复: 不行 实体是一个 行为不同

1

Allen
2021-07-17

为什么不是通过ContactRepo, ReaderRepo, BuyerRepo获取每个上下文的聚合根, 而是通过UserRepo去承担这个职责呢？

展开

作者回复: 聚合根是实体 不是角色

赵晏龙
2021-07-06

到了角色一步，我觉得我遇到的问题基本就解决了，因为我会把【角色】作为【聚合根】使用，可以解释为：在特定上下文中，Buyer、Reader、Contact就是聚合根，User只是一个在它们之间起到关联作用的类，只在【关联关系】相关逻辑上下文中进行使用。我觉得这样建模能够更进一步的将模型解耦。
所以我并没有很好的理解，我这样的解决方案是否还存在老师您讲的后面这些问题。

展开

作者回复: 可以 扮演角色 有不同的实现方法

1

箭和方糖
2021-07-05

又是一个特别好的模式解决一个常见的问题，赞

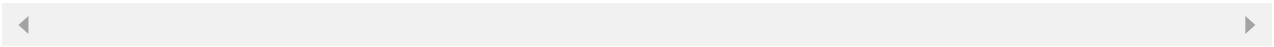
有一个地方不太理解，上下文对象放在UserRepo的好处是什么？可不可以是standalone的？这两个哪个更好

作者回复: repo里可以注入技术上下文

威
2021-07-05

在最后一个赠送课程的例子里，是不是可以用领域服务这概念去包装赠送的逻辑呢，因为从另一个角度看，赠送的业务逻辑，也的确是需要通过多个上下文协作完成，好像放领域服务里也合适

作者回复: 我也不喜欢游离于上下文之外的服务 那不过是过程的另一个名字 不够面向对象

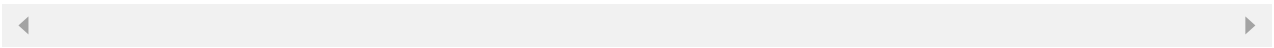


狩月
2021-07-05

在实现中加入上下文依赖， 这样是不是隐藏了“必须得是friend才能赠送订阅”这个领域知识？ 毕竟从领域层已经看不到这个逻辑了， 业务逻辑相当于下放了？

展开

作者回复: 然而模型表达不了规则 怎么也要到代码里

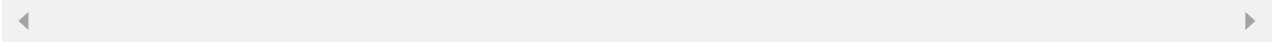


koofrank
2021-07-05

// 订单上下文 public void placeOrder(Column column) { ... }

这个是不是应该返回一个 Order 呢, 然后 repo 去 save, domain 这一层这个实现里面不可以调用外部吧

作者回复: 实现里随便 可参考下一课



koofrank
2021-07-05

有没有上述解决方案后的一个大致实现和目录结构呢, 没有看到放在 github 上面

作者回复: 没

威
2021-07-05

如果针对不同的上下文，在各自上下文的内部都新建一个名为user的类，不同上下文中的user类的字段和方法不尽相同。这种方式可行吗？

展开

作者回复: 不太行 因为在统一语言里 你也不会说 什么什么上下文中的user这样

梦倚栏杆
2021-07-05

有代码仓库吗，可以提供一下吗？

展开

作者回复: 没

冯
2021-07-05

妙！

展开

lan_cyl
2021-07-04

上下文对象中加入依赖，甚至在Reader中依赖其他的上下文，是不是也不太好？另外在模型中怎么体现这种依赖关系呢？
或者是否应该建立新的上下文，例如转赠上下文，来避免上下文之间的耦合呢？

作者回复: 上下文依赖是在实现层面上的 接口层面没有依赖。所以从模型视角 依赖是不存在的

2

webmin

2021-07-04

老师在上一节课和这节课中都提到了smalltalk，今天从Github上找到了一个开源多平台的版本，Run起来感受了一下System Browser，看了一下Collections - Arrayed实现，单看每个词都能懂，但是整段代码怎么干的就完全看不懂了。

有兴趣的同学可以自己下载来去玩一下https://github.com/Cuis-Smalltalk/Cuis-Small...

展开

四仔

2021-07-04

讲得真透彻！

展开

超

2021-07-03

有个疑问，系统一般行为都会跟用户有关，这样建模的话user不就会越来越多内容吗？我还是没法想象具体做系统的时候这个工程怎么实现。

展开

作者回复: 这篇不就是讲怎么做

5